1. Tempo disponibile 120 minuti (90 minuti per gli studenti di "Introduzione agli Algoritmi" - 6 CFU, che devono fare solo i primi 3 esercizi).
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY assumendo implementazione della struttura UnionFind tramite quickUnion con euristica "by rank":

Algorithm 1: MYSTERY(INT n) $\rightarrow$ INT

```
UNIONFIND uf ← new UNIONFIND()
for j ← 1 to n do
    uf.makeSet(j)
end
INT u ← 1
INT v ← n
while u ≤ n do
    uf.union(uf.find(u), uf.find(v))
    u ← u + u
    v ← v/2
end
return uf.find(1)
```

Soluzione. Nel caso di implementazione tramite quickUnion con euristica "by rank", le operazioni di *makeSet* e *union* hanno costo costante, mentre le operazioni di $find$ hanno costo logaritmico rispetto alla dimensione dell'insieme su cui viene effettuata l'operazione stessa. Il ciclo **for** iniziale esegue n operazioni *makeSet*, contribuisce quindi al costo computazionale per una quantità $\Theta(n)$. Le operazioni di assegnamento hanno costo costante. Il **while** viene eseguito $\Theta(logn)$ volte in quanto l'indice u viene raddoppiato ad ogni ciclo. Il costo del corpo del ciclo risulta essere $O(\log n)$ in quanto sono presenti operazioni a costo costante, più due esecuzioni di $find$ su insiemi che sono sicuramente superiormente limitati da n. Complessivamente il **while** contribuisce al costo computazionale con una quantità $O(\log n \times \log n)$. Infine, l'operazione **return** esegue una $find$ con costo $O(\log n)$. Sommando tutti questi contributi avremo che il costo computazionale della funzione MYSTERY risulta essere $T(n) = \Theta(n) + O(1) + O(\log n \times \log n) + O(\log n) = \Theta(n)$. Quest'ultimo passaggio è giustificato dal fatto che $\log n \times \log n = O(n)$; questo può essere compreso ricordando dalle slide del corso che $\log n = O(n^{\frac{1}{2}})$ e quindi anche $(\log n)^2 = O((n^{\frac{1}{2}})^2)$, da cui appunto segue $\log n \times \log n = O(n)$.

2. Si scriva un algoritmo che preso in input un **albero binario** T ritorni TRUE se T è *perfetto* e FALSE in caso contrario. Discutere la complessità della soluzione proposta. Nota: un albero binario è *perfetto* se 1) ogni nodo che non sia una foglia ha esattamente due figli; 2) tutte le foglie sono alla stessa profondità.

Soluzione. Dato un albero binario T, calcoliamo innanzitutto la profondità del nodo più a sinistra che non abbia un figlio sinistro (Algorithmo 2).

Algorithm 2: LeftDepth(Tree T) → Int

```
if T = NULL then
 | return 0
else if T.isLeaf() then
 | return 1
else
 | return 1 + LeftDepth(T.left)
end
```

Notiamo che se l'albero T è perfetto la profondità del nodo più a sinistra coincide con la profondità di tutte le foglie di T. Conoscendo tale profondità possiamo facilmente verificare le condizioni 1 e 2 che caratterizzano un albero perfetto (Algoritmo 3).

Algorithm 3: Perfect(Tree T, Int *depth*) → Bool

```
if T.isLeaf() then
 | return depth = 0
else if T.left = NULL or T.right = NULL then
 | return FALSE
else
 | return Perfect(T.left,depth-1) and Perfect(T.right,depth-1)
end
```

Mettendo insieme gli algoritmi 2 e 3 otteniamo la soluzione al nostro problema (Algoritmo 4). Assumiamo che un albero vuoto sia perfetto.

Algorithm 4: PerfectTree(Tree T) → Bool

```
if T = NULL then
 | return TRUE
else
 | return Perfect(T,LeftDepth(T))
end
```

La funzione LeftDepth (Algoritmo 2) nel caso pessimo attraversa il percorso più lungo radice-foglia, quindi ha un costo O(h), dove h è l'altezza dell'albero, mentre nel caso ottimo (quando la radice non ha figli sinistri) ha un costo costante O(1). La funzione Perfect (Algoritmo 3) nel caso pessimo deve visitare tutti i nodi dell'albero per raggiungere le foglie, quindi ha un costo O(n), dove n è il numero di nodi nell'albero, mentre nel caso ottimo termina immediatamente (costo costante). Poichè $h \leq n$, concludiamo che PerfectTree (Algoritmo 4) ha nel caso pessimo un costo pari a O(n) O(1) nel caso ottimo (ad esempio, albero la cui radice non abbia nessun figlio sinistro).

3. Una cava contiene delle pietre, ognuna con un proprio peso. Bisogna consegnare ad un cliente un preciso peso complessivo di pietre, e quindi si deve capire se è possibile raggiungere esattamente tale peso selezionando alcune delle pietre disponibili. Per risolvere tale problema, progettare un algoritmo che, dati un array di numeri interi non negativi $A[1..n]$, ed un numero intero non negativo K, restituisce *true* se esiste un sottoinsieme dei numeri in A che sommati danno K. Ad esempio, dato $A = [10, 5, 4, 8, 21]$ e $K = 13$, restituisce *true* in quanto $5 + 8 = 13$.

   Soluzione. Il problema richiede di verificare se esiste un sottoinsieme di valori presi da un array di interi $A[1..n]$ che sommati restituiscono un dato valore intero K. Tale problema può essere risolto utilizzando

Cont.

la programmazione dinamica, considerando i seguenti problemi $P(i,j)$, con $i \in \{1..n\}$ e $j \in \{0..K\}$, così definiti:

> $P(i,j) = true$ se esiste un sottoinsieme di elementi presi tra i primi i elementi dell'array A che sommati danno j, altrimenti $P(i,j) = false$.

I problemi $P(i,j)$ possono essere risolti in modo iterativo rispetto all'indice i tenendo in considerazione che:

$$P(i,j) = \begin{cases} true & \text{se } j = 0 \\ true & \text{se } j > 0,\ i = 1 \text{ e } A[1] = j \\ false & \text{se } j > 0,\ i = 1 \text{ e } A[1] \neq j \\ P(i-1,j) & \text{se } j > 0,\ i > 1 \text{ e } A[i] > j \\ P(i-1,j) \text{ or } P(i-1,j-A[i]) & \text{se } j > 0,\ i > 1 \text{ e } A[i] \leq j \end{cases}$$

La soluzione al problema iniziale coincide con $P(n,K)$, ovvero il sottoproblema che considera tutti i valori in ingresso e la somma richiesta risulta essere K.

L'Algoritmo 5 risolve tutti i problemi $P(i,j)$ salvando le relative soluzioni in una matrice booleana B, e alla fine restituisce $B[n,K]$. Il costo computazionale risulta essere $\Theta(n \times K)$ in quanto vengono eseguite alcune operazioni di costo costante per ogni cella della matrice B, avente dimensione $n \times (K+1)$, ma $\Theta(n \times (K+1)) = \Theta(n \times K) + \Theta(n) = \Theta(n \times K)$.

Algorithm 5: PIETRE(INT $A[1..n]$, INT K) → BOOLEAN

```
BOOLEAN B[1..n, 0..K]
for i ← 1 to n do
    B[i,0] ← true
end
for j ← 1 to K do
    if A[1] = j then
        B[1,j] ← true
    else
        B[1,j] ← false
    end
end
for i ← 2 to n do
    for j ← 1 to K do
        if A[i] > j then
            B[i,j] ← B[i-1,j]
        else
            B[i,j] ← B[i-1,j] or B[i-1,j-A[i]]
        end
    end
end
return B[n,K]
```

4. Dato un grafo orientato aciclico $G = (V,E)$, stampare tutti i vertici appartenenti a V in modo tale che se dal vertice v_i è possibile raggiungere il vertice v_j, allora v_i viene stampato prima di v_j.

   Soluzione. Ciò che viene richiesto può essere ottenuto con una stampa dei vertici del grafo secondo un ordinamento topologico. Infatti, l'esistenza di un cammino da v_i a v_j implica l'esistenza di una sequenza di archi (v_i, v_{i+1}), (v_{i+1}, v_{i+2}), $\ldots$, (v_{i+l}, v_j) che implica che, in un ordinamento topologico, v_i apparirà prima di v_j. Come descritto a lezione, un ordinamento topologico può essere ottenuto considerando i vertici in ordine inverso di chiusura secondo una visita in profondità.

   L'Algoritmo 6 effettua una visita DFS del grafo che inserisce i vertici visitati in una struttura LIFO, al momento della chiusura della loro visita. Al termine della visita, i vertici vengono stampati secondo l'ordine

Cont.

di estrazione da tale struttura LIFO. Il costo computazionale di tale algoritmo è il medesimo della visita DFS, quindi $O(n+m)$ (con n numero di nodi e m numero di vertici, assumendo implementazione tramite liste di adiacenza) in quanto le operazioni che sono state aggiunte all'algoritmo DFS sono, per ogni vertice, un inserimento, una lettura ed una cancellazione dallo stack. Queste operazioni contribuiscono al costo computazionale con una quantità $O(n)$, che è trascurabile rispetto al costo $O(n+m)$ della DFS.

Algorithm 6: OrindeTopologico(Graph (E, V))

```
 // Inizializzazione stack e marcatura dei vertici
Stack s ← new Stack ()
for v ∈ V do
 | v.visited ← false
end
 // Esecuzione della DFS
for v ∈ V do
 | if not v.visited then
 | | DFSvisit(v)
 | end
end
 // Estrazione dei vertici dallo stack e stampa
while not s.isEmpty() do
 | print s.top()
 | s.pop()
end

 // DFS che inserisce i vertici nello stack al momento della chiusura
DFSvisit(Vertex u)
u.visited ← true
for v ∈ u.adjacents do
 | if not v.visited then
 | | DFSvisit(v)
 | end
end
s.push(u)
```

The End.